Python Code

Your Name

December 20, 2024

1 Task 1: Image Patches

1.1 Python Code of image patches and 3 patches plots

You can include your Python code here. For example:

```
1
2  def image_patches(image, patch_size=(16, 16)):
3      """
4      Given an input image and patch_size,
5      return the corresponding image patches made
6      by dividing up the image into patch_size sections.
7
8      Input- image: H x W
9             patch_size: a scalar tuple M, N
10     Output- results: a list of images of size M x N
11     """
12     # TODO: Use slicing to complete the function
13     output = []
14     height, width = image.shape[:2]
15     patch_size_height, patch_size_width  = patch_size
16
17     for i in range (0,height, patch_size_height):
18         for j in range (0, width, patch_size_width):
19             patch = image[i:i+patch_size_height, j:j+patch_size_width]
20             patch_mean = np.mean(patch)
21             patch_std = np.std(patch)
22             normalized_patch = (patch - patch_mean) / patch_std
23             output.append(normalized_patch)
24
25
26     return output
```

1.2 why you think it is good for the patches to have zero mean.

- Normalization, Normalization brings all pixel values to a common scale, making the patches more invariant to changes in overall brightness or contrast. This can improve the robustness of algorithms to varying lighting conditions.
- Consistent Representation: helps in achieving a consistent representation across different images.It ensures that the overall brightness level of an image does not dominate the feature extraction process, allowing the algorithm to focus more on the spatial patterns and structures within the patches.
- making them less sensitive to variations in lighting conditions.

1.3 why the patches from the previous question would be good or bad for things like matching or recognizing an object. Consider how those patches would look like if we changed the object's pose, scale, illumination

- Patches with zero mean, obtained through normalization, are advantageous for tasks like matching or recognizing objects in computer vision. Normalized patches provide a consistent representation, making them more robust to changes in pose, scale, and illumination. This ensures that the underlying spatial patterns and features within the patches remain relatively invariant, enhancing the reliability of object matching or recognition across varying conditions.

2 Convolution and Gaussian Filter

2.1 Show $G_{xy} \equiv G_x * G_y$

Gaussian filter $G \in \mathbb{R}^{k\times k}$, Seperating G to two 1D Gaussian filters $G_y \in \mathbb{R}^{k\times 1}$ and $G_x \in \mathbb{R}^{1\times k}$

Let $x[n,m]$ represent the original image. Let $y[n,m]$ be the image after the filter.

When using 2D filter: $(x * G)[m,n] = \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} G[k-i-1, k-j-1]\, x[m+i, n+j]$

Since $G = G_x G_y$ So $= \sum_{i=0}^{k-1}\sum_{j=0}^{k-1} G_x[k-i-1]\, G_y[k-j-1]\, x[m+i, n+j]$

$= \underbrace{\sum_{i=0}^{k-1} G_x[k-i-1]}_{\text{part ①}} \underbrace{\sum_{j=0}^{k-1} G_y[k-j-1]\, x[m+i, n+j]}_{\text{part ②}}$

For part ②: $\sum_{j=0}^{k-1} G_y[k-j-1]\, x[m+i, n+j]$ i doesn't change here. So we can consider it as a

$= (x[m+i][n] * G_y)$ 1D convolution.

Add part ① to part ②: $\sum_{i=0}^{k-1} G_x[k-i-1]\,(x[m+i][n] * G_y)$

$= G_x * x[m][n] * G_y$

$= x[m][n] * G_x * G_y$

So we prove $x[m,n] * G = x[m,n] * G_x * G_y$.

ALSO since $G_x * G_y = G_x G_y$

So $G_x * G_y = \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{x^2}{2\sigma^2}) \cdot \frac{1}{\sqrt{2\pi\sigma^2}} \exp(-\frac{y^2}{2\sigma^2})$

$= \frac{1}{2\pi\sigma^2} \exp(-\frac{x^2+y^2}{2\sigma^2})$

which is equal to $G \in \mathbb{R}^{k\times k}$

2.2 Python code of convolve()

```python
def convolve(image, kernel):
"""
Return the convolution result: image * kernel.
Reminder to implement convolution and not cross-correlation!
Caution: Please use zero-padding.

Input- image: H x W
       kernel: h x w
Output- convolve: H x W
"""

H, W = image.shape
h, w = kernel.shape

#zero padding
h_pad = h // 2
w_pad = w // 2

padded_image = np.pad(image,((h_pad,h_pad),(w_pad, w_pad)), mode = "constant")

#initialize the output image
output_img = np.zeros_like(image)

```

```
    #flip the kernel
    flipped_kernel = np.flipud(np.fliplr(kernel))

    #loop through the image and multiply the kernel with the iamge
    for i in range(H):
        for j in range(W):
            cur_region = padded_image[i:i+h,j:j+w]
            output_img[i,j] = np.sum(cur_region * flipped_kernel)



    return output_img
```

2.3 Load the image "grace_hopper.png" as the input and apply a Gaussian filter & what Gaussian filtering does to the image

- the image:

- what Gaussian filtering does to the image:
  Gaussian filtering smoothens the image by reducing high-frequency noise and details while preserving the overall structure and edges.

2.4 why it is a good idea for a smoothing filter to sum up to 1.

If we don't do the normalization, then it may introduce unintended changes in intensity and produce sharper, potentially more pronounced edges at the expense of increased noise amplification. So in order to maintain the image's overall intensity and produce a more consistent smoothing effect, we should let the smoothing filter sum up to 1.

2.5 the convolution kernels for derivatives:

2.5

i. $k_x \in \mathbb{R}^{1\times 3}$: $I_x = I * k_x$ $\quad k_x$: $[1, 0, -1]$ Since it's convolution, So k_x and k_y

$I_x = [I(x-1, y) \; I(x, y) \; I(x+1, y)] * [1, 0, -1]$ to be flipped.

$= I(x+1, y) - I(x-1, y)$

ii. $k_y \in \mathbb{R}^{3\times 1}$: $I_y = I * k_y$ $\quad k_y$: $[1, 0, -1]^T$

$I_y = [I(x, y-1) \; I(x, y) \; I(x, y+1)]^T * [1, 0, -1]^T$

$= I(x, y+1) - I(x, y-1)$

2.6 python code of edge_detection()

```
def edge_detection(image):
"""
Return Ix, Iy and the gradient magnitude of the input image

Input- image: H x W
Output- Ix, Iy, grad_magnitude: H x W
"""
# TODO: Fix kx, ky
kx = np.array([[-1,0,1]])
ky = np.array([[-1],[0],[1]])

Ix = convolve(image, kx)
Iy = convolve(image, ky)

# TODO: Use Ix, Iy to calculate grad_magnitude
grad_magnitude = np.sqrt(Ix**2 + Iy**2)

return Ix, Iy, grad_magnitude
```

2.7 Plot both outputs and put them in your report & why smoothing an image before applying edge-detection is beneficial. & How would the strength of the smoothing affect the final results?

- Input is the original:

- Input is the image after gaussian filter:

Q: why smoothing an image before applying edge-detection is beneficial:

A: Smoothing an image reduces noise and irregularities so that it will be easier to detect true edges in the image since the noise can introduce false edges and reduce the effectiveness of edge detection algorithms. At the same time, smoothing helps to blur sharp transitions in intensity, which can result in more coherent and continuous edge maps. It helps to suppress high-frequency noise and small-scale structures, leading to more robust edge detection results.

Q: How would the strength of the smoothing affect the final results?

A: If the strength of the smoothing is too high, it will blur the edges and loss details in the original image. Even if it can be beneficial for removing noise and highlighting larger-scale structures or contours in the image. However, it may also lead to over-smoothing and loss of important edge information. But if the strength of the smoothing is too low, it will retain more noise which will affect the accuracy of the edge detection by including false edges, even if it may preserve finer details.

2.8 python code for bilateral_filter() & Compare the results against the Gaussian filter with the same configuration

- The python code of bilateral_filter()

```
        def bilateral_filter(image, window_size, sigma_d, sigma_r):
    """
    Return filtered image using a bilateral filter

    Input-  image: H x W
            window_size: (h, w)
            sigma_d: sigma for the spatial kernel
            sigma_r: sigma for the range kernel
    Output- output: filtered image
    """
    # TODO: complete the bilateral filtering, assuming spatial and range kernels are gaussian
    H, W = image.shape
    h, w = window_size
    print ("H: ", H)
    print ("W: ", W)
    print ("h: ", h)
    print ("w: ", w)
    pad_h = h // 2
    pad_w = w // 2
    #create the padded image
    padded_image = np.pad(image, ((pad_h,pad_h), (pad_w, pad_w)),mode="constant", constant_values=0)
    print("padded_image shape is: ", padded_image.shape)
    output_image = np.zeros_like(image)
    for i in range(H):
        for j in range(W):
            i_min = i
            i_max = i+h
            j_min = j
            j_max = j+w
            #initialize the spatial kernel
```

```
            spatial_kernel = np.zeros((h,w))
            #generate the value in each element of the kernel
            for x in range(h):
                for y in range(w):
                    spatial_kernel[x,y] = np.exp(-((x-pad_h)**2 +(y-pad_w)**2 )/(2*sigma_d**2))
            #generate the range kernel
            intensity_diff = padded_image[i_min:i_max, j_min:j_max] - padded_image[i,j]
            range_kernel = np.exp(-intensity_diff**2 / (2*sigma_r**2))

            #combine two kernels together
            combined_kernel = spatial_kernel * range_kernel

            # print("spatial_kernel is: ", spatial_kernel)

            # print ("combined_kernel is: ", combined_kernel)

            #normalize the kernal
            combined_kernel /= np.sum(combined_kernel)
            # print("each sum is: ", np.sum(combined_kernel))
            #use the combined kernel to the image
            output_image[i,j] = np.sum(padded_image[i_min:i_max, j_min:j_max] * combined_kernel )



    return output_image

```

- Compare the results against the Gaussian filter with the same configuration
  Bilateral filter output:

  Gaussian filter output:

3 Sobel Operator

3.1 If the input image is I and we use G_s as our Gaussian filter, taking the horizontal-derivative of the Gaussian-filtered image, can be approximated by applying the Sobel filter

3.1 k_x : $[-1\ 0\ 1]$

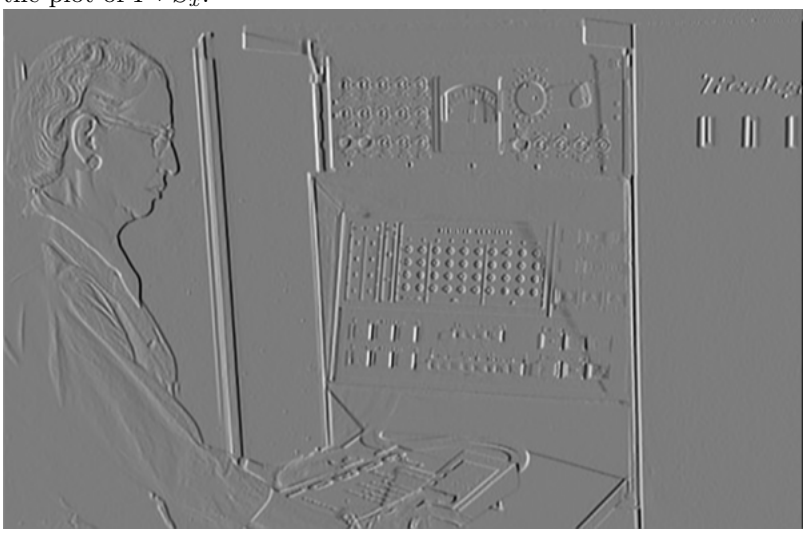
$$G_s * k_x : \begin{bmatrix} 0 & 1 & 2 & 1 & 0 \\ 0 & 2 & 4 & 2 & 0 \\ 0 & 1 & 2 & 1 & 0 \end{bmatrix} * [-1\ 0\ 1] = \begin{bmatrix} 2 & 0 & -2 \\ 4 & 0 & -4 \\ 2 & 0 & -2 \end{bmatrix} \div 2 = \begin{bmatrix} 1 & 0 & -1 \\ 2 & 0 & -2 \\ 1 & 0 & -1 \end{bmatrix} = S_x$$

3.2 python code of sobel_operator()

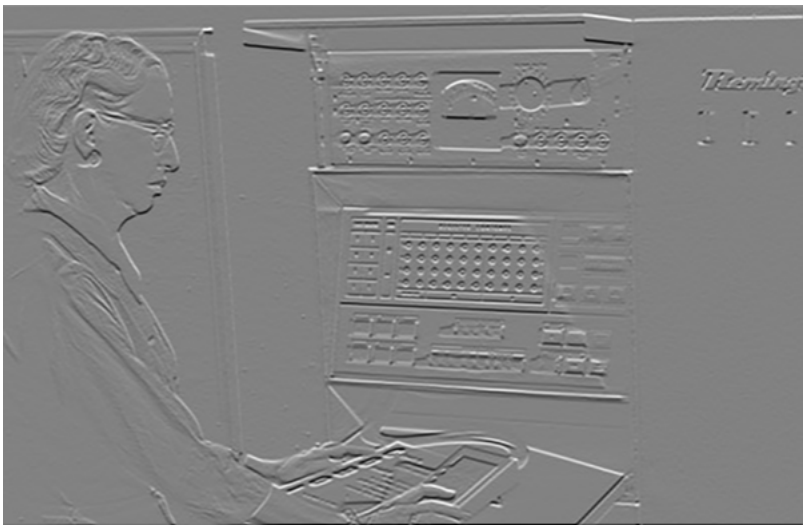
```
def sobel_operator(image):
"""
Return Gx, Gy, and the gradient magnitude.

Input- image: H x W
Output- Gx, Gy, grad_magnitude: H x W
"""
# TODO: Use convolve() to complete the function
Sx = np.array([[1,0,-1],[2,0,-2],[1,0,-1]])
Sy = np.array([[1,2,1],[0,0,0],[-1,-2,-1]])
Gx = convolve(image,Sx)
Gy = convolve(image, Sy)
grad_magnitude = np.sqrt(Gx**2 + Gy**2)
return Gx, Gy, grad_magnitude
```

3.3 the plots of $I * S_x, I * S_y$, gradient magnitude

the plot of $I * S_x$:



the plot of $I * S_y$:

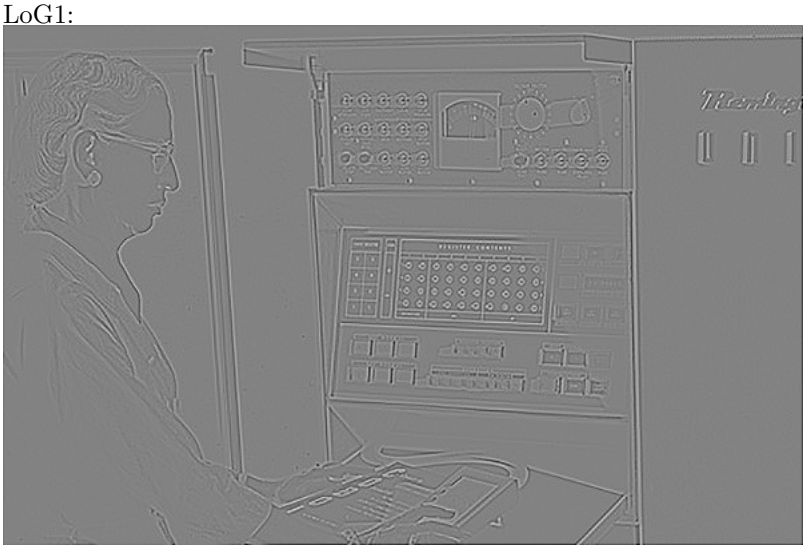


the plot of gradient magnitude:

4 LoG Filter

4.1 the outputs of these two LoG filters & the reasons for their difference.

LoG1:



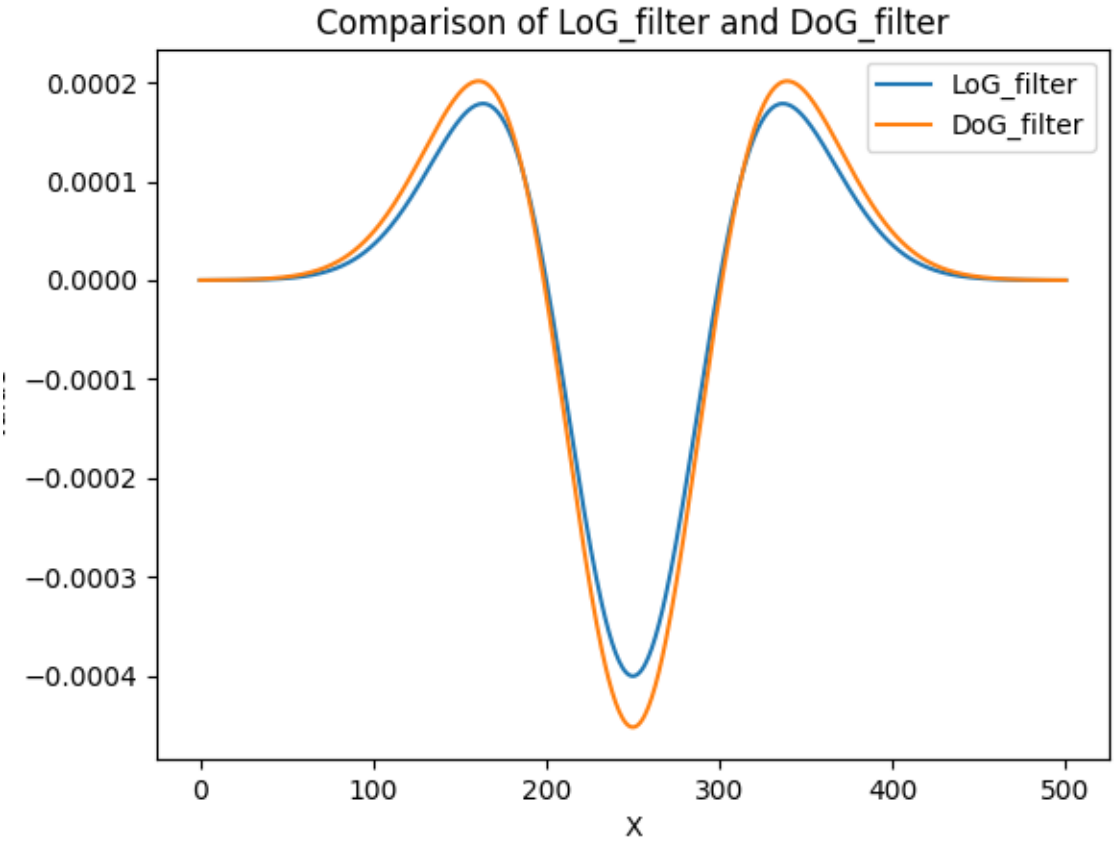
LoG2:

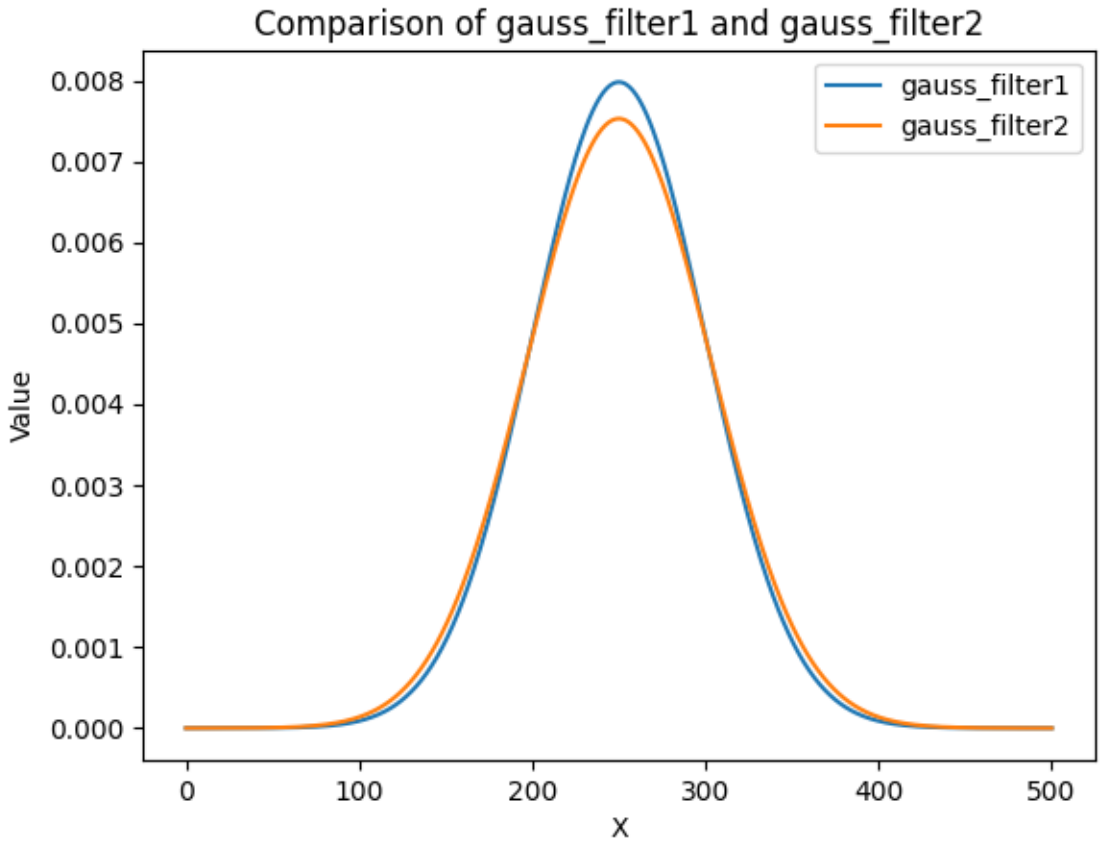
Q: the reasons for their difference.

A: The reason for there difference is the first filter is 3*3 and the second filter is 9 * 9. Due to its smaller size and simpler structure, Kernel_LoG1 might provide a smoother response, emphasizing larger-scale features and edges. Kernel_LoG2, with its larger size and more complex structure, can capture more localized features and variations in intensity, providing a more detailed response.

4.2 Discuss in your report why computing $(I * G_{k_\sigma}) - (I * G_\sigma)$ might successfully roughly approximate convolution by the Laplacian of Gaussian. You should include a plot or two showing filters in your report.

Two plots:





Q: Explanation:

A: since convolution is linear, so if the DoG (G1 - G2) is roughly equal to LoG, then we can use the DoG to replace the LoG and save the calculation time. From the second plot, we can see the LoG and DoG are roughly the same.

5 Who's That Filter?

5.1 Write out each of the four remaining filters. No need to format them prettily

- $[[1,1,1],[1,1,1],[1,1,1]]$
- $[[1/9,1/9,1/9],[1/9,1/9,1/9],[1/9,1/9,1/9]]$
- $[[0,0,0],[1,0,0],[0,0,0]]$
- $[[-1,0,1],[-1,0,1],[-1,0,1]]$

5.2 What does filter 1 do, intuitively & how does it differ from filter 2?

Q: What does filter 1 do

A: Filter 1, intuitively, performs a basic all operation known as a blur. It replaces each pixel's value with the sum of value of the pixel's neighborhood, including the pixel itself. This process effectively blurs the image, but increases the intensity of the picture.

Q: how does it differ from filter 2

A: The main difference between Filter 1 and Filter 2 lies in how they handle the normalization of the filter coefficients. In Filter 1, there is no normalization applied. However, filter 2 applies normalization by dividing each element of the filter by the sum of all elements in the filter. This ensures that the total contribution of the filter to each pixel's value is consistent and that the brightness of the image is preserved. By normalizing the filter coefficients, Filter 2 maintains the overall brightness of the image while still achieving the blurring effect.

6 Corner Score

6.1 Python code of corner_score()

```
    def corner_score(image, u=5, v=5, window_size=(5, 5)):
    """
    Given an input image, x_offset, y_offset, and window_size,
    return the function E(u,v) for window size W
    corner detector score for that pixel.
    Use zero-padding to handle window values outside of the image.

    Input- image: H x W
           u: a scalar for x offset
           v: a scalar for y offset
           window_size: a tuple for window size

    Output- results: a image of size H x W
    """
     # Offset the image by (u,v)
    offset_image = np.roll(image, (u,v), axis=(0, 1))

    # Take the squared difference with the original image
    squared_diffs = (image - offset_image) ** 2

    # Compute the sum of squared differences within the window using convolution
    kernel = np.ones(window_size)
    sum_squared_diffs = scipy.ndimage.convolve(squared_diffs, kernel, mode="constant", cval=0)

    return sum_squared_diffs
```

6.2 your output for for grace_hopper.png for $(u, v) = \{(0, 5), (0, -5), (5, 0), (-5, 0)\}$ and window size $(5, 5)$

6.3 why checking all the us and vs might be impractical in a few sentences.

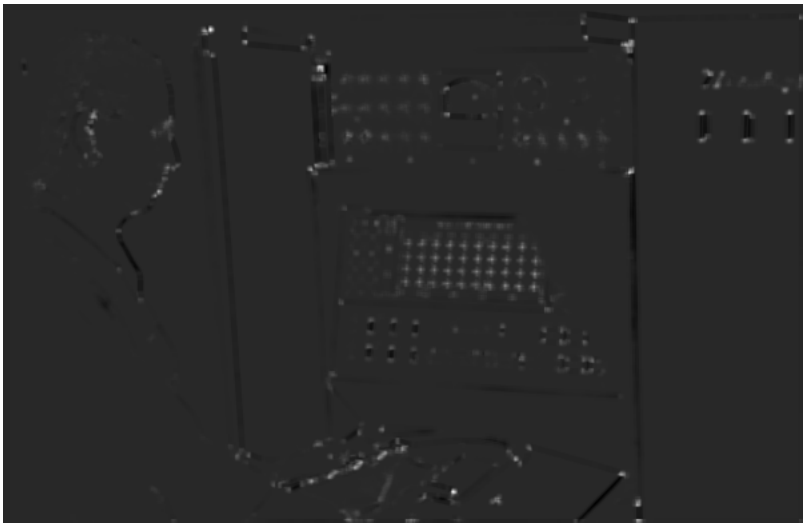
The corner score computation involves comparing pixel intensities within a window, which requires nested loops over the window size and the image dimensions. With larger images and window sizes, the number of comparisons grows significantly, leading to a quadratic increase in computation time. Therefore, exhaustively checking all possible u and v could become computationally prohibitive.

7 Harris Corner Detector

7.1 python code of harris_detector()

```
def harris_detector(image, window_size=(5, 5)):
"""
Given an input image, calculate the Harris Detector score for all pixels
You can use same-padding for intensity (or 0-padding for derivatives)
to handle window values outside of the image.

Input- image: H x W
Output- results: a image of size H x W
"""

x_filter = np.array([[-1,0,1]])
y_filter = np.array([[-1],[0],[1]])

# compute the derivatives
Ix = scipy.ndimage.convolve(image,x_filter,mode="constant", cval=0)
Iy = scipy.ndimage.convolve(image,y_filter,mode="constant", cval=0)

Ixx = Ix**2
Ixx = scipy.ndimage.convolve(Ixx, np.ones(window_size), mode="constant", cval=0)
Iyy = Iy**2
Iyy = scipy.ndimage.convolve(Iyy, np.ones(window_size), mode="constant", cval=0)
Ixy = Ix * Iy
Ixy = scipy.ndimage.convolve(Ixy, np.ones(window_size), mode="constant", cval=0)

det_M = Ixx * Iyy - Ixy**2
trace_M = Ixx + Iyy

arpha = 0.04
R = det_M - arpha * trace_M**2
# For each image location, construct the structure tensor and calculate
# the Harris response


return R
```

7.2 plot Harris Corner Detector score for every point



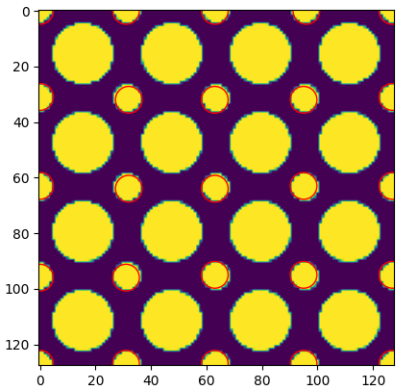
8 Single-Scale Blob Detection

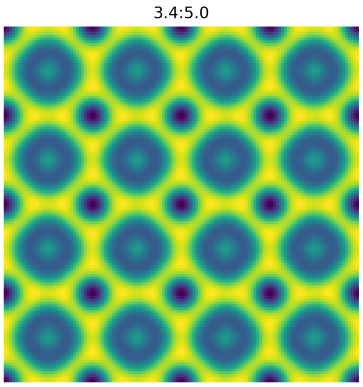
8.1 python code of gaussian_detection.py

```
    def gaussian_filter(image, sigma):
    """
    Given an image, apply a Gaussian filter with the input kernel size
    and standard deviation

    Input
      image: image of size HxW
      sigma: scalar standard deviation of Gaussian Kernel

    Output
      Gaussian filtered image of size HxW
    """
    H, W = image.shape
    # -- good heuristic way of setting kernel size
    kernel_size = int(2 * np.ceil(2 * sigma) + 1)
    # Ensure that the kernel size isn't too big and is odd
    kernel_size = min(kernel_size, min(H, W) // 2)
    if kernel_size % 2 == 0:
        kernel_size = kernel_size + 1
    # TODO implement gaussian filtering of size kernel_size x kernel_size
    # Similar to Corner detection, use scipy's convolution function.
    # Again, be consistent with the settings (mode = 'reflect').

    G_kernel = np.zeros((kernel_size, kernel_size))
    center = kernel_size // 2
    for i in range (kernel_size):
        for j in range (kernel_size):
            x = i - center
            y = j - center
            curr_value =(1 / (2 * np.pi*np.square(sigma))) * np.exp(-(np.square(x) + np.square(y))/(2*np.
    square(sigma)))
            G_kernel[i,j] = curr_value

    output = scipy.ndimage.convolve(image, G_kernel, mode="reflect")
    return output
```

8.2 two responses and report the parameters used to obtain each & how many maxima are you observing? & are there false peaks that are getting high values?

small polka:





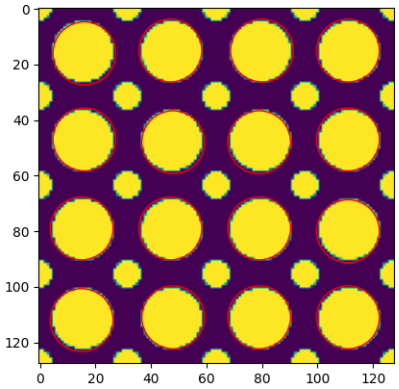
Q: how many maxima are you observing?

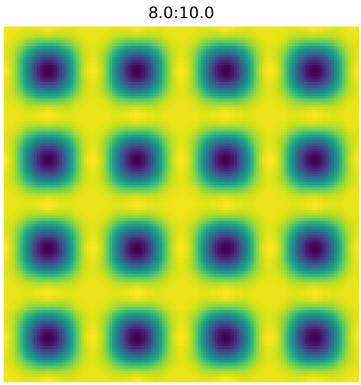
A: $16 + 25 = 41$

Q: Are there false peaks that are getting high values?

A: yes, we change the sigma1 and sigma2 so that i should only pick the small circle, however, from the maxima visualization plot, we see the blob detection program also consider the edges of the big polka as the small polka and generate the false peak.

large polka:





Q: how many maxima are you observing?

A: 16

Q: Are there false peaks that are getting high values?

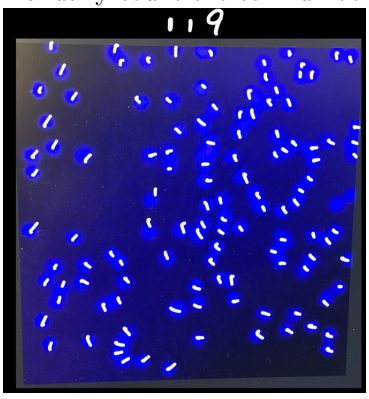
A: no, when we use bigger sigma1 and sigma2 it will only pick the larger polka.

9 Cell Counting

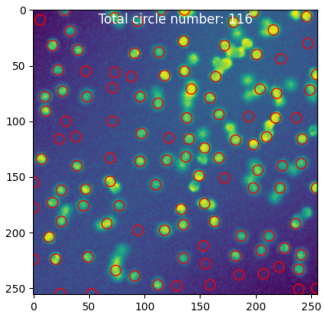
9.1 a set of parameters for generating the scale space and finding the maxima that allows you to accurately detect the cells in each of those images.

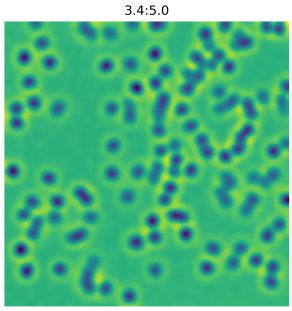
- without any reprocessing & using sigma1 = 3.3 sigma2 = 5

result: use cell_001 as the resource.
manually count the cell number is 119



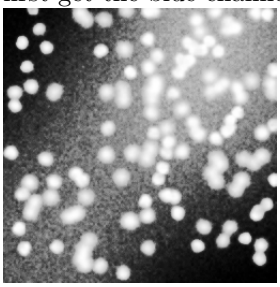
there are total 116 cell detected.



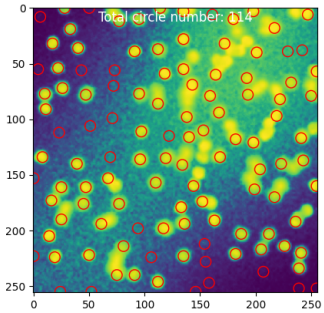


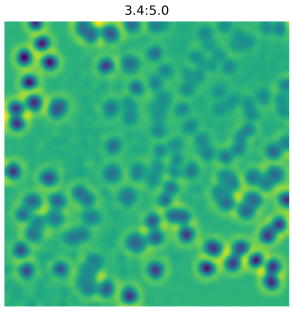
the wrong circle number is 31
the correct circle number is 80
the correct detection rate is 67%

- using the sharpening kernel to preprocess the image & keep the sigma1 and sigma2 be the same.
first get the blue channel from the cell image, and the sharpen it:



use this as the input to the blob detection program:





there are total 114 cell detected.
the wrong circle number is 24
the correct circle number is 90
the correct detection rate is 75%

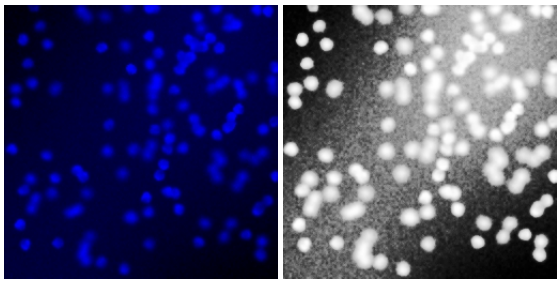
9.2 python code for cell detection and preprocessing

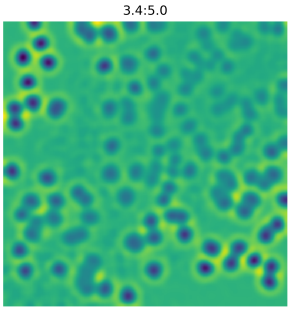
```
    # # # -- TODO Task 9: Cell Counting --

  # # Apply sharpening filter
  # image = cv2.imread('cells/001cell.png')
  # blue_channel = image[:, :, 0]



  # equalized_image = cv2.equalizeHist(blue_channel)

  # # Save or display the black and white image
  # cv2.imwrite('./cell_detections/black_white_image.png', blue_channel)
  # cv2.imwrite('./cell_detections/sharpen.png', equalized_image)


```

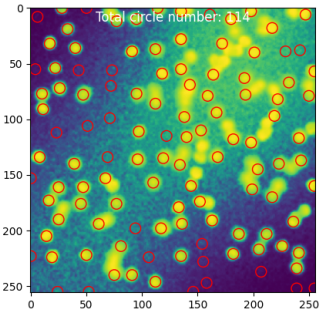
```
    print("Detecting cells")

    # Detect the cells in any four (or more) images from vgg_cells
    # Create directory for cell_detections
    if not os.path.exists("./cell_detections"):
        os.makedirs("./cell_detections")

    # cell_image = read_img('cells/001cell.png')
    cell_image = read_img('cell_detections/sharpen.png')
    sigma_1, sigma_2 = 3.4, 5
    gauss_1 = gaussian_filter(cell_image, sigma_1)  # to implement
    gauss_2 = gaussian_filter(cell_image, sigma_2)   # to implement

    # calculate difference of gaussians
    DoG_small = gauss_2 - gauss_1  # to implement

    # visualize maxima
    maxima = find_maxima(DoG_small, k_xy=10)
    visualize_scale_space(DoG_small, sigma_1, sigma_2 / sigma_1,
                          './cell_detections/cell_DoG_with_preprocess.png')

    # visualize_maxima(cell_image, maxima, sigma_1, sigma_2 / sigma_1,
    #                  './cell_detections/cell_circle_3.4.png')
```

after we find the proper sigma1 and sigma2 value and the proper preprocessing method, we will then choose other 3 cell image as the input

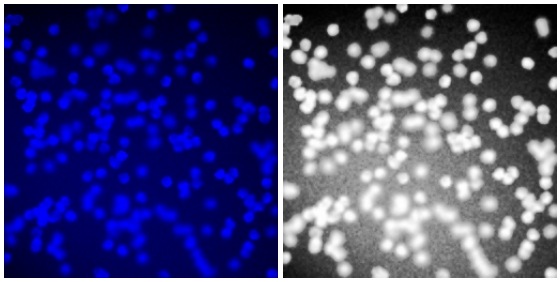
- cell_001

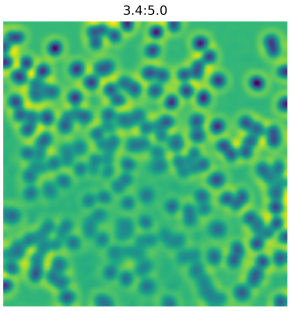


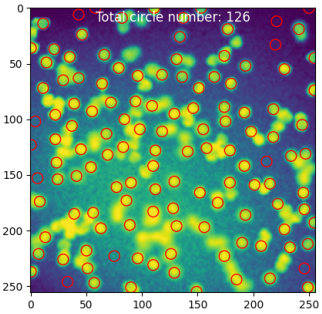




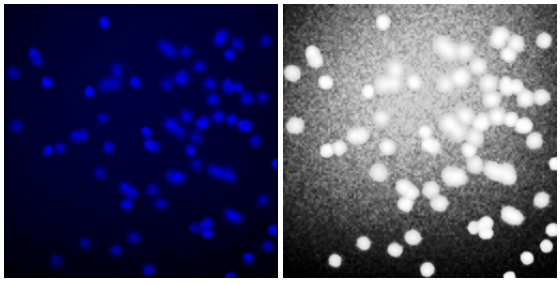
- cell_002

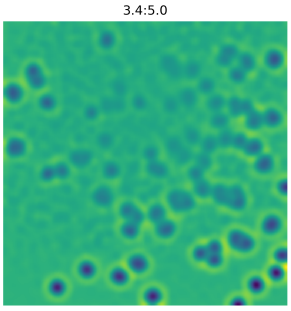


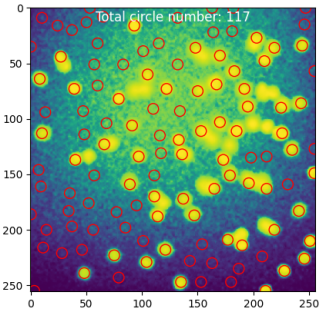




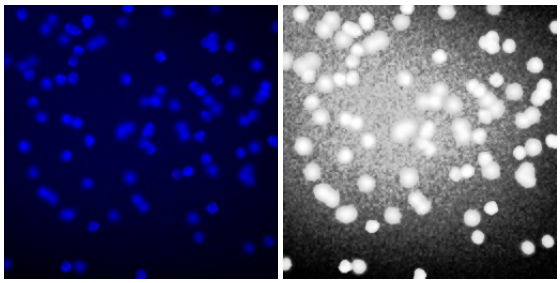
- cell_004

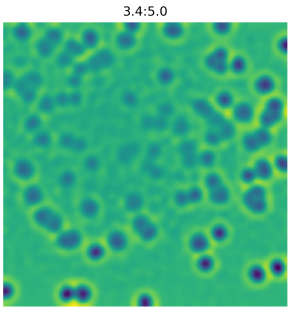


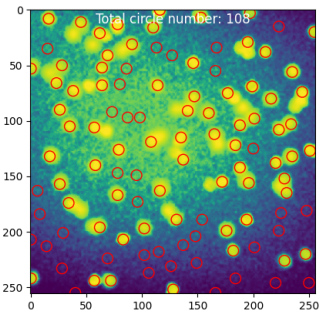




- cell_005




3.4:5.0


Total circle number: 108
0
50
100
150
200
250
0
50
100
150
200
250